File Handling and Memory Management

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). Since, random access memory **(RAM)** is volatile which loses its data when computer is turned off, we use files for future use of the data. When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file

2. Read or write (perform operation)

3. Close the file

1. How to open a file?

Python has a built-in function **open()** to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the **open()** function. The **open()** function takes two parameters; filename, and mode.

There are different methods (modes) for opening a file:

i. **"r"** - Read - Default value. Opens a file for reading, error if the file does not exist.
ii. **"a"** - Append - Opens a file for appending, creates the file if it does not exist
iii. **"w"** - Write - Opens a file for writing, creates the file if it does not exist
iv. **"x" -** Create - Creates the specified file, returns an error if the file exists
v. **"r+" -** To read and write data into the file. The previous data in the file will be overridden.
vi. **"w+" -** To write and read data. It will override existing data.
vii. **"a+" -** To append and read data from the file. It won't override existing data.

In addition, you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

'+' Open a file for updating (reading and writing)

1.1 Opening file in read format:

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

or

```
f=open (“demofile.txt”, ‘r’)
```

The code above is the same as:

```
f = open ("demofile.txt", "rt")
```

Because **"r"** for read, and **"t"** for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

If demofile.txt exists and let us assume the contents are:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in **open()** function.

The **open()** function returns a file object, which has a **read()** method for reading the content of the file:

a. **Example for reading all whole file at a time**

```
f = open("demofile.txt", "r")
print(f.read())
```

output:

```
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!
```

b. Example for Reading Only Parts of the File

By default the **read()** method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")

print(f.read(5))
```

Output:

```
Hello
```

c. Example to Read Lines

You can return one line by using the **readline()** method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")

print(f.readline())
```

By calling **readline()** two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")

print(f.readline())

print(f.readline())
```

Output:

```
Hello! Welcome to demofile.txt

This file is for testing purposes.
```

d. **By looping through the lines of the file, you can read the whole file, line by line:**

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
        print(x)
```

output:

```
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!
```

1.2. Opening a file in append format

To append in to a file we must open file with append mode and we can add more data with existing data and if that particular file does not exist it creates new file and we can enter data in to that file.

To print the data after appending data we have to close file after appending and then open file again in read mode

Example:

```
f=open('demofile.txt','a')
f.write("this is kiranmaie")
f.close()
f=open('demofile.txt','r')
print(f.read())
f.close()
```

Output:

```
Hello! Welcome to demofile.txt
```

```
This file is for testing purposes.

Good Luck!

This is kiranmaie
```

1.3 Opening a file in write mode:

To write in to file open file with write mode if the file is already existed with data that data will be lost and it will be empty we can enter new data in to it if file does not exist it creates new file with that name and we can enter data in to it to display data in that file after write mode close the file and open again with read mode

Example:

```
f=open('demofile.txt','w')

f.write("oops all data deleted")

f.close()

f=open('demo.txt' ,'r')

print(f.readline(5))

f.close()
```

1.4 Opening a file in r+ mode:

r+ mode is read and write mode we can open a file in read mode and we can write data in to it with out closing the file and reopen in write mode

Example:

```
f=open('siri.txt' , 'r+')

for each in f:

    print(each)

print()

f.write("data was over written")

print(f.read())

f.close()
```

1.5 Opening a file in w+ mode:

To write and read data. It will override existing data.

Example:

```
with open("siri.txt","w+") as f:
    f.write("kiranmaie")
```

1.6 Opening file in a+ mode:

To append and read data from the file. It won’t override existing data.

Example:

```
with open("siri.txt", "a+") as f:
    f.write("love u")
```

2. Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
```

To Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

To Delete a File

To delete a file, you must import the OS module, and run its **os.remove()** function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, then delete it:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the **os.rmdir()** method:

Example

Remove the folder "myfolder":

```
import os

os.rmdir("myfolder")
```

Note: You can only remove empty folders.

We can change our current file cursor (position) using the **seek()** method. Similarly, the **tell()** method returns our current position (in number of bytes).

```
>>> f.tell() # get the current file position

56

>>> f.seek(0) # bring file cursor to initial position

0
```

Reading and Writing CSV File using Python

SV (stands for comma separated values) format is a commonly used data format used by spreadsheets. The csv module in Python's standard library presents classes and methods to perform read/write operations on CSV files.

writer()

This function in csv module returns a writer object that converts data into a delimited string and stores in a file object. The function needs a file object with write permission as a parameter. Every row written in the file issues a newline character. To prevent additional space between lines, newline parameter is set to ‘’.

The writer class has following methods

writerow()

This function writes items in an iterable (list, tuple or string) ,separating them nby comma character.

writerows()

This function takes a list of iterables as parameter and writes each item as a comma separated line of items in the file.

Following example shows use of write() function. First a file is opened in ‘w’ mode. This file is used to obtain writer object. Each tuple in list of tuples is then written to file using writerow() method.

```
>>> import csv

>>> persons=[('Lata',22,45),('Anil',21,56),('John',20,60)]

>>> csvfile=open('persons.csv','w', newline='')

>>> obj=csv.writer(csvfile)

>>> for person in persons:
```

```
obj.writerow(person)
>>> csvfile.close()
```

This will create ‘persons.csv’ file in current directory. It will show following data.

```
Lata,22,45
Anil,21,56
John,20,60
```

Instead of iterating over the list to write each row individually, we can use writerows() method.

```
>>> csvfile = open('persons.csv','w', newline='')
>>> obj = csv.writer(csvfile)
>>> obj.writerows(persons)
>>> obj.close()
```

read()

this function returns a reader object which returns an iterator of lines in the csv file. Using the regular for loop, all lines in the file are displayed in following example.

```
>>> csvfile=open('persons.csv','r', newline='')
>>> obj=csv.reader(csvfile)
>>> for row in obj:
print (row)
['Lata', '22', '45']
['Anil', '21', '56']
['John', '20', '60']
```

Since reader object is an iterator, built-in next() function is also useful to display all lines in csv file.

```
>>> csvfile = open('persons.csv','r', newline='')
>>> obj = csv.reader(csvfile)
>>> while True:
try:
row=next(obj)
print (row)
except StopIteration:
break
```

The csv module also defines a dialect class. Dialect is set of standards used to implement CSV protocol. The list of dialects available can be obtained by list_dialects() function.

```
>>> csv.list_dialects()
['excel', 'excel-tab', 'unix']
```

DictWriter()

This function returns a DictWriter object. It is similar to writer object, but the rows are mapped to dictionary object. The function needs a file object with write permission and a list of keys used in dictionary as fieldnames parameter. This is used to write first line in the file as header.

writeheader()

This method writes list of keys in dictionary as a comma separated line as first line in the file.

In following example, a list of dictionary items is defined. Each item in the list is a dictionary. Using writrows() method, they are written to file in comma separated manner.

```
>>> persons=[{'name':'Lata', 'age':22, 'marks':45}, {'name':'Anil', 'age':21, 'marks':56}, {'name':'John', 'age':20, 'marks':60}]
>>> csvfile=open('persons.csv','w', newline='')
>>> fields=list(persons[0].keys())
>>> obj=csv.DictWriter(csvfile, fieldnames=fields)
>>> obj.writeheader()
>>> obj.writerows(persons)
>>> csvfile.close()
```

The file shows following contents.

```
name,age,marks
Lata,22,45
Anil,21,56
John,20,60
```

DictReader()

This function returns a DictReader object from the underlying CSV file. As in case of reader object, this one is also an iterator, using which contents of the file are retrieved.

```
>>> csvfile = open('persons.csv','r', newline='')
>>> obj = csv.DictReader(csvfile)
```

The class provides fieldnames attribute, returning the dictionary keys used as header of file.

```
>>> obj.fieldnames
['name', 'age', 'marks']
```

Use loop over the DictReader object to fetch individual dictionary objects.

```
>>> for row in obj:
print (row)
```

This results in following output.

```
OrderedDict([('name', 'Lata'), ('age', '22'), ('marks', '45')])
OrderedDict([('name', 'Anil'), ('age', '21'), ('marks', '56')])
OrderedDict([('name', 'John'), ('age', '20'), ('marks', '60')])
```

To convert OrderedDict object to normal dictionary, we have to first import OrderedDict from collections module.

```
>>> from collections import OrderedDict
>>> r=OrderedDict([('name', 'Lata'), ('age', '22'), ('marks', '45')])
>>> dict(r)
{'name': 'Lata', 'age': '22', 'marks': '45'}
```

Memory Management Operations.

What is memory management?

Memory management in simple terms means, the process of providing memory required for your program to store data and freeing up unused data in memory is called memory management.

Providing memory is called **memory allocation**. Freeing up memory is called **memory de-allocation.**

In Python , **Memory manager** is responsible for allocating and de-allocating memory.

Why memory management is required?

Generally, programming languages uses objects to operate on data required by your program. These objects are created in-memory for faster access. So , once an object is created it is allocated some space on memory , once your program completes its execution these objects has to be clean up or deleted from the memory because they are no longer in use which can re-used again for other processes/program for execution.

If these un-used objects are not cleaned up, then your memory might be full and there will not be enough space for other programs and your application might crash. So, memory management is very important in any programming language.

In early programming languages (like C), it was developers responsibility to allocate memory and de-allocate memory after the execution which led to below problems:

***Forgetting to free up memory* —** *If developer forgets to free up un-used memory, then memory might become full which leads to your program using too much memory.*

***Freeing memory which is already in use* —** *If developer by mistake frees up memory which is already in use, which causes an issue when your program tries to access same memory which results in unexpected behavior.*

So, these problems led new programming languages to implement automatic **Memory Management** and **Garbage Collection** which is taken care by **Python Memory Manager** in Python.

In python memory allocation and de-allocation is automatic.

How memory is allocated in Python?

So, as said in **python memory manager** is responsible for **allocating and de-allocating the memory.**

Memory has two parts stack memory and heap memory(which has nothing to do with heap data structure)

Stack Memory —all the methods/method calls , references are store in stack memory.

Heap Memory — all the objects are stored in heap.

Everything in python is an Object. So, it is very important to understand about objects in python.

Python is a dynamically typed language, which means types are assigned based on the value it is referring to unlike other programming languages(Java and C#).

Unlike in other programming languages , in python whenever a variable is assigned a value , the python memory manager will check if an object with that value is already available in the memory . If object is already present in memory , then this variable points to that object instead of creating a new object with the same value.

If object with that value is not available in memory(heap) , python memory manager will create a new object with the specified value and this variable will point to this newly created object on heap(memory).

Also , when a variable is re-assigned with new variable , instead of overriding the value in memory , what python does is , it will again follow same process as above and checks if there is an object already present on heap with the new value . If object is already present , then this variable will point to that object or else python memory manager will create a new object on heap with new value and this variable will point to that value.

For example,

x == 100 // this will create a new object in heap
y == 100 // this will not create a new object as an object with value 100 is already available on heap
print(id(x) == id(y)) // this returns true because x and y are pointing to same object on heap
x = 101 // now when new value is assigned "101" is not available on heap , so new object is created and x points to this new object . In this case value at that location is not overwritten unlike other programming languages.

Which is not the case with other programming languages where when a variable is updated, the value at that memory location/address is overwritten with new update value

Whenever a new object is created in python, python memory manager ensures that there is enough memory in the heap to allocate space to that object.

In python, all objects are derived from **PyObject a struct** which has two properties reference count and pointer to the object

Garbage Collection in Python

Now, it is time to clean up the objects which are not in use. The process of de-allocating the memory or deleting the un-used objects so that it can be made available to other objects is called **Garbage Collection**.

So , the job of the garbage collector is to track of the objects which can de-allocated.

Python uses below 2 algorithms for garbage collection:

- Reference counting

- Generational garbage collection

Reference Counting

Reference counting is a simple technique in which, whenever the reference count of an object reaches to “0”, then it is eligible for garbage collection and the memory allocated for that object is automatically de-allocated.

Whenever an object is created, the reference count of that object is incremented by “1” and similarly, when a reference to that object is removed, then its reference count is decremented by “1”. Finally, when the reference count of that object becomes “0”, the memory allocated for that object is de-allocated.

Python “sys” module, provides a method called “getrefcount(object)” ,which gives the count of references for a given object.

By default python uses reference counting technique for garbage collection, which cannot be disabled (developer has no control over it).

But the issue with this technique is it has some overhead because, every object has to keep track of reference count for memory de-allocation and also memory de-allocation happens whenever an objects reference count becomes “0”.

Reference counting will not be able to detect the cyclic references and those objects will not be eligible for garbage collection.

Because of above problems, python also uses another technique called **Generational Garbage Collection.**

Generational Garbage Collection

This is also an automatic process in python, but unlike reference counting which cannot be disabled; generational garbage collection is optional and can also be triggered manually.

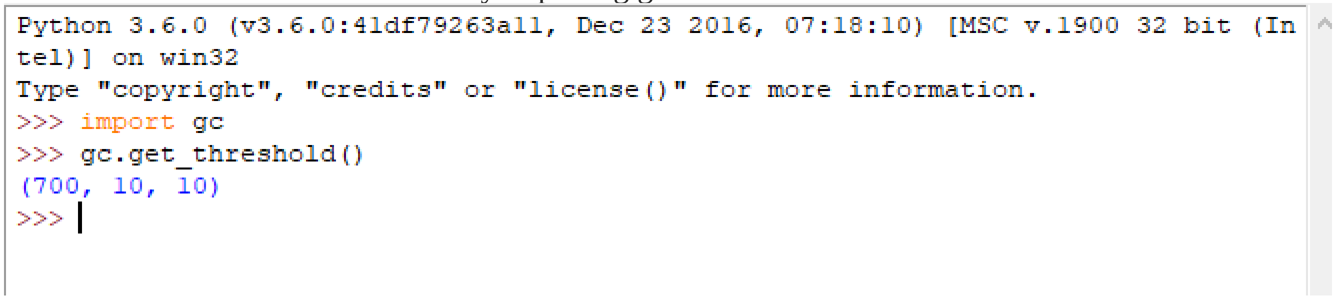
gc module in python is responsible for generational garbage collection.

In this technique, all python objects are classified into 3 categories:

- Generation 0
- Generation 1
- Generation 2

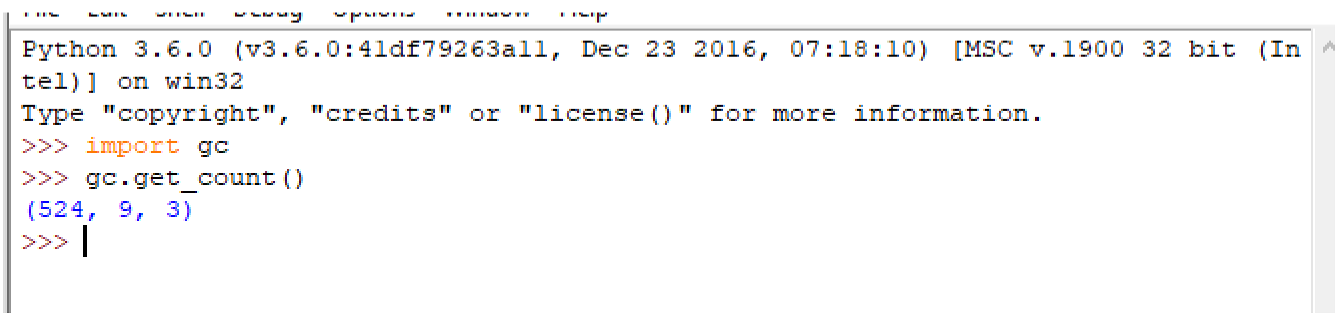
Each generation has a predefined **threshold**, threshold is nothing but it is an indicator for garbage collector on when to invoke garbage collection.

You can check the default threshold by importing gc module as below:



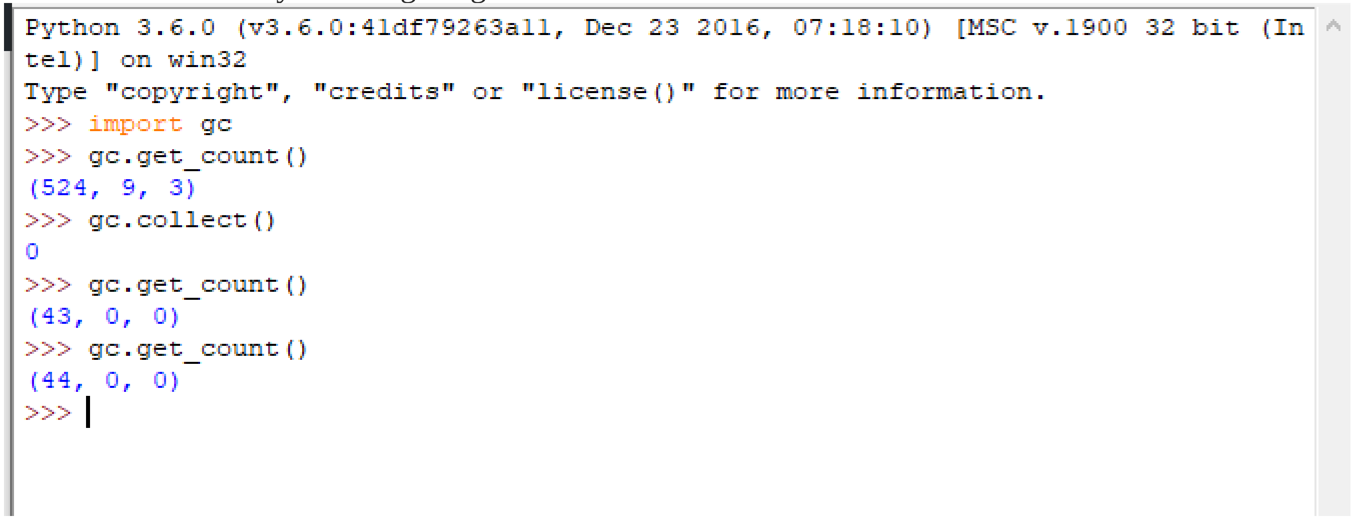
```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import gc
>>> gc.get_threshold()
(700, 10, 10)
>>> 
```

You can also check the number of objects in each generation as below:



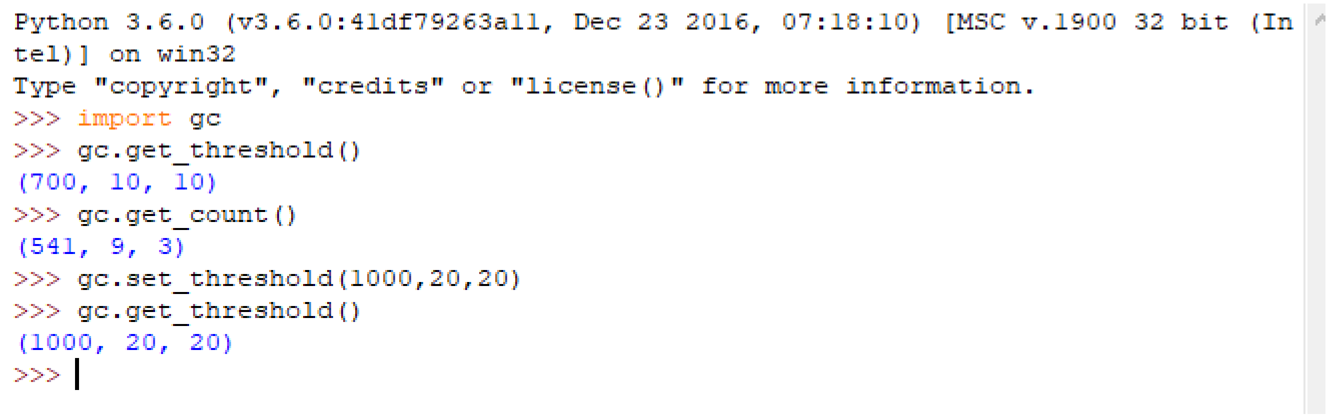
```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import gc
>>> gc.get_count()
(524, 9, 3)
>>> 
```

You can also manually call the garbage collection as below:



```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import gc
>>> gc.get_count()
(524, 9, 3)
>>> gc.collect()
0
>>> gc.get_count()
(43, 0, 0)
>>> gc.get_count()
(44, 0, 0)
>>> 
```

You can also set the threshold as below:



```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import gc
>>> gc.get_threshold()
(700, 10, 10)
>>> gc.get_count()
(541, 9, 3)
>>> gc.set_threshold(1000,20,20)
>>> gc.get_threshold()
(1000, 20, 20)
>>> 
```

When a new object is created, that object is categorized into “generation 0”.

Garbage collection is triggered automatically when a generation reaches its threshold and whatever objects remain in that generation after garbage collection is promoted to older generation.

If there are 2 generations reached threshold, always garbage collection chooses older generation and then younger generation.